A proposed categorical semantics for Pure ML*

Wesley Phoa and Michael Fourman
Laboratory for the Foundations of Computer Science
James Clerk Maxwell Building, The King's Buildings
Edinburgh EH9 3JZ, Scotland

Abstract

This paper outlines an approach to modelling the purely functional fragment of ML: it concentrates on the semantics of the Modules system. Our proposed semantics is *set-theoretic:* types and values are modelled by sets and functions in a *topos*, a categorical model of constructive set theory. *Synthetic domain theory* allows us to make sense of fixed points and recursive domains in a set-theoretic setting, while the notions of *classifying topos* and 'generic' structure provide a useful way of interpreting signatures, functors and sharing, as well as Extended ML specifications. We only give an informal account, concentrating on motivation and examples rather than giving a rigorous formal development—only elementary category theory is used.

1 Introduction

The process of specifying the formal semantics of Standard ML involved an unusually close interplay between theoretical and practical considerations. The official Definition, [12], was finalised in 1989. But a denotational semantics would be useful, e.g. for reasoning about programs. Indeed, a draft semantics (by Don Sannella) helped guide the development of the language; but it never reached a definitive form.

A model of ML should be clean and intuitive; the different aspects of the language should be treated in a unified framework; it should be modular; and we should be able to extend it easily to handle more sophisticated languages. Finally, it should legitimise familiar, informal, methods of reasoning.

This paper sketches an approach to modelling ML, focussing on the Modules system. There are at least four ways of interpreting Modules:

- *Operational:* following the official Definition of Standard ML: [12], [2], [21];
- *Type-theoretic:* treating signatures as specifying existential types: [13], [3];
- *Algebraic:* identifying a signature with its class of models: [19], [7]; and
- *Categorical:* using the notion of generic structure, as proposed here.

*Research supported by SERC grant GR/F 31199, 'Formal System Design'; the first author was also supported by an 1851 Research Fellowship

Though our categorical methods look superficially different from the other approaches, they are closely related to the algebraic viewpoint; in fact, some of our proposals formally resemble ideas from algebraic specification, though the setting we work in is more suited to (higher-order) program logic. There are also connections with the type-theoretic view, not fully understood. But our approach was more directly inspired by the informal picture of the Modules system presented in [2] and [21].

Although we focus on modules, an important feature of our approach is that the different parts of the language fit together in a single, relatively simple framework motivated by set theory; to demonstrate this, we have included a brief discussion of the semantics of Core ML.

Our work is based on some rather technical results, but no knowledge of these will be required: only some familiarity with categories. In fact, the presentation is quite informal. The precise details are being worked out; for example, the use of classifying toposes to model ML polymorphism is described in detail in [17] (in the more general framework of *polynomial categories*).

History and acknowledgements The idea of using classifying toposes to interpret ML polymorphism and modules was proposed by Fourman some years ago, and independently by Phoa in 1991; this paper is the first place in which it has been worked out. Walters (with his student Lack) has been developing similar ideas in the context of imperative languages based on distributive categories.

Phoa was supported by an 1851 Research Fellowship. He would also like to thank Bob Walters, Mike Johnson, James McKinna, Dave Berry and especially Ed Kazmierczak for stimulating discussions, and Carl Gunter, Philippa Gardner and the members of the ML club at the LFCS for their encouragement. Healfdene Goguen and John Power patiently read drafts and made valuable comments. So did the referees.

2 The semantics of Core ML

2.1 Constructive set theory and toposes

Our original intuition is that data types, and more complicated semantic domains, are *sets*. But for technical reasons—the nonexistence of fixed points, recursive types, polymorphic types and so on—we must work, not with sets, but with domains. This can be inconvenient: for example, we must always check that functions are continuous. It is also not clear how far we can reason naively, or set-theoretically, about a domain-theoretic semantics. It does seem clear that most constructions on domains should be specified abstractly, using *category theory*.

Since the mid-1970s, Scott has been suggesting that set theory can after all provide a basis for denotational semantics: only we must use, not ordinary (classical) set theory, but *constructive* set theory. (In recent years, a lot of progress has been made: e.g. [18] and [15].) We can combine the set-theoretic and categorical viewpoints by working within a categorical formulation of constructive set theory. Such a formulation is provided by *topos theory:* [8].

A topos is a category which 'behaves like' the category of sets: any topos provides an interpretation of constructive set theory, suitably formulated. Though the details of the definition are not needed in this paper, we reproduce it here for completeness:

Definition 2.1 A category $\mathcal{E}$ (with pullbacks) has a **subobject classifier** if there is a "set of truth-values" $\Omega \in \mathcal{E}$ and a map $tt: 1 \to \Omega$ such that for any mono $X' \rightarrowtail X \in \mathcal{E}$, there is a unique map $\chi_{X'}: X \to \Omega$ such that $X' \rightarrowtail X$ is the pullback of tt along $\chi_{X'}$. A category $\mathcal{E}$ is a **topos** if it has finite limits, exponentials and a subobject classifier. ■

The simplest example of a topos is $\mathcal{S}ets$, the usual category of sets. In this paper we will work with $\mathcal{E}\!f\!f$, Hyland's effective topos: [4]. This is the 'universe of recursive mathematics'. For example, Church's thesis, and all the theorems of recursive mathematics, are true in the effective topos.

2.2 Synthetic domains

There are a number of full subcategories of $\mathcal{E}\!f\!f$ whose objects 'behave like domains': for example, see [18], [15], [1]. We will review the necessary results from [15], which is the most naive of these papers; however, nearly all our remarks apply to the other categories as well.

There is a simple way to define a certain full subcategory $\mathcal{CS}ig$ of $\mathcal{E}\!f\!f$, the category of **complete Σ-spaces**. We can regard the objects of $\mathcal{CS}ig$ simply as sets. Some examples of complete Σ-spaces are:

- basic data types like $\mathbb{N}$ and 2 (note that in $\mathcal{E}\!f\!f$, $2 \neq \Omega$)
- products $P \times Q$ and sums $P + Q$, where $P, Q \in \mathcal{CS}ig$
- (total) set-theoretic function spaces $[P \to Q]$
- spaces $[P \rightharpoonup Q]$ of 'computable' partial functions

For example, it contains an object $[\mathbb{N} \to \mathbb{N}]$ whose elements correspond to total recursive functions; and an object $[\mathbb{N} \rightharpoonup \mathbb{N}]$, whose elements 'are' the partial recursive functions. That is, effectivity is built into the model.

The important facts about complete Σ-spaces are:

- if $P \in \mathcal{CS}ig$, then there is a *canonical* partial order on P, under which P has suprema of chains;
- the canonical partial order on a function space $[P \to Q]$ or $[P \rightharpoonup Q]$ coincides with the pointwise order; and
- if $P, Q \in \mathcal{CS}ig$, *every* map $P \to Q$ is monotone and preserves suprema of chains.

This canonical ordering has a simple definition motivated by recursion theory.

These results mean that we can regard a complete Σ-space, *qua* set, as a predomain, without having to impose a partial order on it; and domain-theoretic constructions, such as function spaces, coincide with the ordinary set-theoretic constructions. From now on we will simply refer to complete Σ-spaces as **predomains**.

Equality types in ML have a semantic counterpart in our model. A predomain P is an **equality predomain** if

$$\forall p, p' \in P.\, (p = p') \vee (p \neq p')$$

is true in $\mathcal{E}\!f\!f$. This is equivalent to the existence of an equality function $P \times P \to 2$. It is not a vacuous definition because the logic of $\mathcal{E}\!f\!f$ is constructive. For example, $\mathbb{N}$ is an equality predomain; but $[\mathbb{N} \rightharpoonup \mathbb{N}]$ is not, since there is no effective procedure for deciding whether two partial recursive functions are equal.

From now on we write $\mathcal{S} = \mathcal{E}\!f\!f$ to emphasise that nothing which follows depends on the fact that we are working with the effective topos $\mathcal{E}\!f\!f$; it all works more generally. See [16] for other examples of suitable toposes.

2.3 Semantics of the Core

There are two aspects of ML which we will not consider here. Firstly, we are not concerned with references and exceptions. Secondly, we do not deal with type reconstruction. So we are really working with XML, a pure, explicitly typed variant of SML: see [13].

The way we interpret Core ML without polymorphic values is standard. Following Plotkin and [20], we do not use domains (with a bottom element) but predomains and partial function spaces. For example

$$[\![\texttt{nat}]\!] = \mathbb{N}, \qquad [\![\sigma \texttt{ * } \tau]\!] = [\![\sigma]\!] \times [\![\tau]\!], \qquad [\![\sigma \texttt{ -> } \tau]\!] = [[\![\sigma]\!] \rightharpoonup [\![\tau]\!]]$$

Datatypes are interpreted by recursive domains—domain equations can be solved in the usual way. For example, $[\![\texttt{nat list}]\!]$ is the solution, in the category of predomains, of the domain equation

$$X \cong 1 + (\mathbb{N} \times X)$$

In the case of 'algebraic' datatypes such as lists, for which it makes sense to solve this equation set-theoretically, it can be shown that the domain-theoretic solution of this equation coincides with the set-theoretic solution. (This applies, not just to 'flat' predomains, but to predomains in general.) One consequence is that algebraic datatypes constructed from equality predomains are again equality predomains, as ML requires. Another consequence is that the naive unrestricted induction principles for algebraic datatypes are, by definition, sound.

Once we have defined the semantics of closed types using predomains, ML polymorphism can be added transparently: we use *polynomial categories* over $\mathcal{S}$, as in [17]. In this case they are just the classifying toposes to be defined in the next section: a value whose type contains a type variable is interpreted in the topos $\mathcal{S}[X]$, not $\mathcal{S}$. For example, the polymorphic identity `fn x => x :  'a -> 'a` is interpreted by the arrow

$$\ulcorner 1_X \urcorner : 1 \to [X \rightharpoonup X] \in \mathcal{S}[X]$$

'Real' polymorphic types in $\mathcal{S}$ itself (though they do exist) are not required.

3 Classifying toposes

3.1 Basic definitions

The notion of classifying topos, and of a 'generic' or 'indeterminate' structure of a signature or model of a theory, goes back to the 1970s: see [6, chapter 6]. It can be seen as a special case of a general notion of classifying category. The classifying toposes we'll use are not as general as the ones described in [6]—they can be constructed more directly. However, the following discussion is still slightly oversimplified—for example, we gloss over all the 2-categorical issues that arise.

We will need a notion of morphism between toposes that "preserves all set-theoretic structure and all set-theoretic properties".

Definition 3.1 If $\mathcal{E}, \mathcal{F}$ are toposes, an a**-functor** $f^*: \mathcal{E} \to \mathcal{F}$ is a functor that preserves finite limits, exponentials and the subobject classifier, and has a right adjoint (and hence preserves arbitrary colimits). ■

Suppose $\mathcal{E}, \mathcal{F}$ are constructed out of $\mathcal{S}$ in some way, so that they contain full subcategories of predomains in the same way that $\mathcal{S}$ does. Since constructions on predomains are described set-theoretically, a-functors must preserve all domain theoretic constructions too—such as partial function spaces and solutions of domain equations.

In fact, we will deal with toposes $\mathcal{E}$ that come equipped with an embedding $\mathcal{S} \hookrightarrow \mathcal{E}$ (itself an a-functor). They are constructed by "adding extra objects to $\mathcal{S}$". Such a topos is called an $\mathcal{S}$**-topos**; note that it will contain a full subcategory of predomains, just like $\mathcal{S}$ does. We restrict our attention to toposes equipped with such an embedding, and a-functors $f^*: \mathcal{E} \to \mathcal{F}$ such that

$$\begin{tikzcd} \mathcal{E} \arrow[rr, "f^*"] & & \mathcal{F} \\ & \mathcal{S} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

We will refer to f^* as an a**-functor over** $\mathcal{S}$. Let $\mathcal{S}$**-Top** denote the category of $\mathcal{S}$-toposes and a-functors over $\mathcal{S}$.

We also need to talk about $\mathcal{E}$-toposes and a-functors over $\mathcal{E}$ for $\mathcal{S}$-toposes $\mathcal{E}$; the definitions are just the same.

A typical way to construct an $\mathcal{E}$-topos is by adding an 'indeterminate' or 'generic' predomain X to $\mathcal{S}$ to obtain a new topos $\mathcal{S}[X]$. This will contain all the old objects of $\mathcal{S}$, but also new ones like

$$2 \times [X \rightharpoonup \mathbb{N}], \qquad [\mathbb{N} \rightharpoonup X\ \mathtt{list}], \ldots$$

Intuitively, X is a new predomain about which we know nothing. "Substituting a real predomain $P \in \mathcal{S}$ for X" will give us an a-functor $\mathcal{S}[X] \to \mathcal{S}$ taking

$$X \mapsto P, \qquad 2 \times [X \rightharpoonup \mathbb{N}] \mapsto 2 \times [P \rightharpoonup \mathbb{N}], \ldots$$

The precise definition of $\mathcal{S}[X]$ is categorical:

Definition 3.2 The **polynomial topos (for predomains) over $\mathcal{S}$** is a topos $\mathcal{S}[X]$ such that $\mathcal{S} \hookrightarrow \mathcal{S}[X]$, containing a distinguished predomain $X \in \mathcal{S}[X]$, the **generic predomain**, and having the following universal property:

- For every $\mathcal{S}$-topos $\mathcal{E}$ and for every predomain $P \in \mathcal{E}$ there is a unique a-functor $\mathcal{S}[X] \to \mathcal{E}$ over $\mathcal{S}$ taking $X \mapsto P$.

$\mathcal{S}[X]$ is said to **classify** predomains in $\mathcal{S}$, and in $\mathcal{S}$-toposes. ∎

Note that we can adjoin more than one indeterminate predomain to $\mathcal{S}$, to obtain

$$\mathcal{S}[X_1, \ldots, X_n]$$

We can also add a generic equality predomain Y to $\mathcal{S}$ to obtain a polynomial topos $\mathcal{S}[Y]$; the definition is nearly the same, and is left to the reader. We don't discuss equality predomains in the rest of the paper, as they do not, in this paper, introduce any additional problems.

It also makes sense to add an indeterminate element x to a predomain P in $\mathcal{S}$ (or some other topos) to obtain a new topos $\mathcal{S}[x : P]$. Again, "substituting any real element $p \in P$ for x" defines an a-functor $\mathcal{S}[x : P] \to \mathcal{S}$ over $\mathcal{S}$ taking $x \mapsto p$.

Before giving the precise definition, note that if $\mathcal{E}$ is an $\mathcal{S}$-topos, any predomain $P \in \mathcal{S}$ can be regarded as a predomain in $\mathcal{E}$ via the embedding; we call it $P \in \mathcal{E}$ again. Similarly if $\mathcal{F}$ is an $\mathcal{E}$-topos and $P \in \mathcal{E}$ is a predomain, we have a predomain $P \in \mathcal{F}$.

Definition 3.3 Given a $\mathcal{S}$-topos $\mathcal{E}$ and a predomain $P \in \mathcal{E}$, the **polynomial topos (for elements of P) over $\mathcal{E}$** is a topos $\mathcal{E}[x : P]$ such that $\mathcal{E} \hookrightarrow \mathcal{E}[X]$, containing a distinguished element $x{:}\,1 \to P$, the **generic element of** P, and having the following universal property:

- For every $\mathcal{E}$-topos $\mathcal{F}$ and for every element $p{:}\,1 \to P \in \mathcal{F}$, there is a unique a-functor $\mathcal{E}[X] \to \mathcal{F}$ over $\mathcal{E}$ taking $x \mapsto p$.

$\mathcal{E}[x : P]$ is said to **classify** elements of P in $\mathcal{E}$-toposes. ∎

We will sketch the constructions. However, they aren't needed in the rest of the paper—we only ever use the universal property of $\mathcal{S}[X]$ or $\mathcal{S}[x : P]$.

Proposition 3.4 *(Cf. [8, ex. 2, p. 211]) Given a predomain $P \in \mathcal{E}$, $\mathcal{E}[x : P]$ is the slice topos $\mathcal{E}/P$. The embedding $\mathcal{E} \hookrightarrow \mathcal{E}/P$ sends $Q \mapsto \left(\begin{smallmatrix} P\times Q \\ \downarrow \pi_0 \\ P \end{smallmatrix}\right)$, and x is the arrow $1 \cong \left(\begin{smallmatrix} P \\ \downarrow 1_P \\ P \end{smallmatrix}\right) \to \left(\begin{smallmatrix} P\times P \\ \downarrow \pi_0 \\ P \end{smallmatrix}\right)$ arising from the diagonal map $\delta{:}\, P \times P \to P$.*

Proposition 3.5 *(Cf. [17, section 4.2]) Recall from [15], [16] that the category of predomains in $\mathcal{S}$ is the* externalisation *(in the sense of [5]) of an internal full subcategory $\mathbb{C} = (C_1 \rightrightarrows C_0)$ in $\mathcal{S}$ arising from a map $f{:}\, G \to C_0$ in $\mathcal{S}$: see [6, example 2.38]. Then $\mathcal{S}[X]$ is $\mathcal{S}/C_0$, where $X = \left(\begin{smallmatrix} G \\ \downarrow f \\ C_0 \end{smallmatrix}\right) \in \mathcal{S}/C_0$.*

A similar construction gives the classifying topos $\mathcal{S}[Y]$ for equality predomains.

4 The semantics of the Modules system

4.1 Signatures without substructure components

The ML modules system allows programmers to define *structures*, which are collections of types, values and substructures. Each structure can match one or more *signatures* or 'interfaces:' lists of names for type components, names/types for value components, and signatures for substructures. There are also *functors*, which take structures of one signature to structures of another. The reader is referred to [9], [2] for more details.—From now on, the term 'a-functor' refers to an a-functor *over* $\mathcal{S}$.

Signatures will be interpreted using classifying toposes. We illustrate this via an example. Consider the signature

```
signature SIGA =
  sig
    type t;
    val f : t -> nat
  end;
```

To give a structure of this signature in our model, we must specify a predomain P and an element $f_0 : 1 \to [P \rightharpoonup \mathbb{N}]$ in $\mathcal{S}$. But this amounts to giving an a-functor out of a polynomial topos:

$$(\mathcal{S}[X])[x : [X \rightharpoonup \mathbb{N}]] \longrightarrow \mathcal{S}$$

We say that the topos

$$[\![\texttt{SIGA}]\!] = (\mathcal{S}[X])[x : [X \rightharpoonup \mathbb{N}]]$$

classifies structures of signature `SIGA`. The particular 'structure'

$$\langle X, x : 1 \to [X \rightharpoonup \mathbb{N}]\rangle$$

in $[\![\texttt{SIGA}]\!]$ is called the **generic** structure of signature `SIGA`.

Proposition 4.1 *The generic structure of signature `SIGA` has the following universal property: given any 'real' structure* `s : SIGA` *in* $\mathcal{S}$*, i.e. any pair* $\langle P \in \mathcal{S}, p \in [P \rightharpoonup \mathbb{N}]\rangle$*, there is a unique a-functor*

$$[\![\texttt{s}]\!] : [\![\texttt{SIGA}]\!] \longrightarrow \mathcal{S}$$

such that $[\![\texttt{s}]\!](X) = P$ *and* $[\![\texttt{s}]\!](x) = p$.

We have assumed that each structure is assigned its principal signature. Actually, signature matching can be modelled using 'instantiation' a-functors between appropriate classifying toposes. The result will generalise to signatures with substructure components. The proof uses only the universal properties of the classifying toposes, and is essentially straightforward.

Proposition 4.2 *If* $\texttt{SIG2} \geq \texttt{SIG1}$ *(see [12, 5.9]) then there is an a-functor*

$$\mathrm{inst}^{\texttt{SIG1}}_{\texttt{SIG2}} : [\![\texttt{SIG1}]\!] \longrightarrow [\![\texttt{SIG2}]\!]$$

Given a structure of signature `SIG2` *(an a-functor* $[\![\texttt{SIG2}]\!] \to \mathcal{S}$*), precomposing with* $\mathrm{inst}^{\texttt{SIG1}}_{\texttt{SIG2}}$ *yields the corresponding structure of signature* `SIG1`.

4.2 A general signature is a cocone of toposes

Recall from [2] that structures and signatures can be visualised as directed acylic graphs or **DAG**s. A signature with no substructure components corresponds to a DAG with just one node. More generally, the signatures

```
signature SIGB =
  sig
    structure ba1 : SIGA;
    structure ba2 : SIGA
  end;
```

```
signature SIGC =
  sig
    structure ca : SIGA;
    structure cb : SIGB
    sharing ca = cb.ba1
  end;
```

correspond to the following DAGs:

$$\begin{tikzcd} & \texttt{SIGB} \arrow[dl, "\texttt{ba1}"'] \arrow[dr, "\texttt{ba2}"] & \\ \texttt{SIGA} & & \texttt{SIGA} \end{tikzcd}$$

$$\begin{tikzcd} \texttt{SIGC} \arrow[dd, "\texttt{ca}"'] \arrow[dr, "\texttt{cb}"] & & \\ & \texttt{SIGB} \arrow[dl, "\texttt{ba1}"] \arrow[dr, "\texttt{ba2}"] & \\ \texttt{SIGA} & & \texttt{SIGA} \end{tikzcd}$$

where we have labelled the vertices with the names of the appropriate signatures and the edges with the substructure component names. Note that these DAGs have a distinguished vertex, the **head**, labelled by the main signature. Any other vertex can be reached from the head. The terminal vertices correspond to signatures without substructure components.

[For simplicity, we assume in this paper that if a signature specifies substructure components, it specifies *only* substructure components, not type or value components. We also assume that sharing specifications involve only components of the same signature, so that no admissification (structure unification) is involved. Both assumptions can be dispensed with: the first, simply by spelling out how type and value identifiers are bound to objects and arrows in toposes; the second, by adding an extra vertex for each sharing specification, labelled by a signature computed from a principal admissifier.]

In our model, the interpretation of a signature `SIG` with DAG G will be a diagram of shape G, *with all the arrows reversed*, in the category $\mathcal{S}$-**Top**. This diagram must be a colimiting cocone, whose head is an $\mathcal{S}$-topos we call

$$[\![\texttt{SIG}]\!]$$

We construct this diagram inductively. For example, we already know what $[\![\texttt{SIGA}]\!]$ is; so we can simply let $[\![\texttt{SIGB}]\!] = [\![\texttt{SIGA}]\!] + [\![\texttt{SIGA}]\!]$. That is, we have a coproduct diagram

$$[\![\texttt{SIGA}]\!] \xrightarrow{[\![\texttt{ba1}]\!]} [\![\texttt{SIGB}]\!] \xleftarrow{[\![\texttt{ba2}]\!]} [\![\texttt{SIGA}]\!]$$

And given this, we can obtain $\llbracket\mathtt{SIGC}\rrbracket$ by forming the finite colimit diagram

$$\begin{tikzcd}
\llbracket\mathtt{SIGC}\rrbracket & & \\
& \llbracket\mathtt{SIGB}\rrbracket \arrow[ul, "\llbracket\mathtt{cb}\rrbracket"'] & \\
\llbracket\mathtt{SIGA}\rrbracket \arrow[uu, "\llbracket\mathtt{ca}\rrbracket"] \arrow[ur, "\llbracket\mathtt{ba1}\rrbracket"'] & & \llbracket\mathtt{SIGA}\rrbracket \arrow[ul, "\llbracket\mathtt{ba2}\rrbracket"']
\end{tikzcd}$$

One can show that $\mathcal{S}$-**Top** has the finite colimits we require. (They are in fact easy to construct: all the toposes $\llbracket\mathtt{SIG}\rrbracket$ turn out to be slice toposes of $\mathcal{S}$, so we simply use the finite limits in $\mathcal{S}$.)

A structure `c : SIGC` is interpreted as an a-functor

$$\llbracket\mathtt{c}\rrbracket\colon \llbracket\mathtt{SIGC}\rrbracket \to \mathcal{S}$$

Because $\llbracket\mathtt{SIGC}\rrbracket$ is a colimit, to construct $\llbracket\mathtt{c}\rrbracket$ it suffices to construct suitable a-functors $\llbracket\mathtt{SIGA}\rrbracket, \llbracket\mathtt{SIGB}\rrbracket \to \mathcal{S}$, i.e. to interpret the components of `c`.

From $\llbracket\mathtt{c}\rrbracket$ we can recover the components of `c`. For example,

$$\llbracket\mathtt{c.ca}\rrbracket = \llbracket\mathtt{c}\rrbracket \circ \llbracket\mathtt{ca}\rrbracket\colon \llbracket\mathtt{SIGA}\rrbracket \longrightarrow \llbracket\mathtt{SIGC}\rrbracket \longrightarrow \mathcal{S}$$

The commutativity of the diagram enforces the sharing requirement

$$\llbracket\mathtt{c.ca}\rrbracket = \llbracket\mathtt{c}\rrbracket \circ \llbracket\mathtt{ca}\rrbracket = \llbracket\mathtt{c}\rrbracket \circ \llbracket\mathtt{cb}\rrbracket \circ \llbracket\mathtt{ba1}\rrbracket = \llbracket\mathtt{c.cb.ba1}\rrbracket$$

Here is an example involving non-local sharing. Suppose we have defined a fixed structure `a : SIGA`, and consider the signature

```
signature SIGD =
  sig
    structure da1 : SIGA;
    structure da2 : SIGA
    sharing da1 = a
  end;
```

This signature corresponds to the DAG

$$\begin{tikzcd}
& \mathtt{SIGD} \arrow[dl, "\mathtt{da1}"'] \arrow[dr, "\mathtt{da2}"] & \\
\mathtt{a : SIGA} & & \mathtt{SIGA}
\end{tikzcd}$$

The topos $\llbracket\mathtt{SIGD}\rrbracket$ appears in the colimit diagram

$$\begin{tikzcd}
& & \llbracket\mathtt{SIGD}\rrbracket & \\
\mathcal{S} \arrow[urr] & \llbracket\mathtt{SIGA}\rrbracket \arrow[l, "\llbracket\mathtt{a}\rrbracket"] \arrow[ur, "\llbracket\mathtt{da1}\rrbracket"'] & & \llbracket\mathtt{SIGA}\rrbracket \arrow[ul, "\llbracket\mathtt{da2}\rrbracket"']
\end{tikzcd}$$

Note that since the diagram commutes, $[\![\texttt{da1}]\!]\colon [\![\texttt{SIGA}]\!] \to [\![\texttt{SIGD}]\!]$ is the composite

$$[\![\texttt{SIGA}]\!] \xrightarrow{[\![\texttt{a}]\!]} \mathcal{S} \hookrightarrow [\![\texttt{SIGD}]\!]$$

If we have a structure `d : SIGD`, i.e. an a-functor

$$[\![\texttt{d}]\!]\colon [\![\texttt{SIGD}]\!] \to \mathcal{S}$$

this forces $[\![\texttt{d.da1}]\!]$ to be $[\![\texttt{a}]\!]$. Again, a rigorous treament awaits a more careful analysis of the binding of names.

It seems possible to extend our approach: for instance, we hope to describe higher-order modules in our framework (note that the labelled DAGs we drew can be parametrised by objects in $\mathcal{S}$). However, other possible extensions of the modules system, such as recursive definitions, seem more problematic.

4.3 A functor is an a-functor in the wrong direction

Again we illustrate the idea using an example. Consider the functor

```
functor F(sb : SIGB) : SIGC =
  struct
    structure ca = sb.ba1;
    structure cb = sb
  end;
```

`F` will be modelled by an a-functor

$$[\![\texttt{F}]\!]\colon [\![\texttt{SIGC}]\!] \longrightarrow [\![\texttt{SIGB}]\!]$$

We define $[\![\texttt{F}]\!]$ by using the fact that $[\![\texttt{SIGC}]\!]$ is the colimit of a cocone.

In more detail, the definition of `F` gives us a-functors

$$\mathrm{id}_{[\![\texttt{SIGB}]\!]} \circ [\![\texttt{ba1}]\!] = [\![\texttt{ba1}]\!] \circ \mathrm{id}_{[\![\texttt{SIGA}]\!]}\colon [\![\texttt{SIGA}]\!] \longrightarrow [\![\texttt{SIGB}]\!]$$

$$\mathrm{id}_{[\![\texttt{SIGB}]\!]}\colon [\![\texttt{SIGB}]\!] \longrightarrow [\![\texttt{SIGB}]\!]$$

which are compatible and so give us a cocone with vertex $[\![\texttt{SIGB}]\!]$; this in turn induces the required a-functor $[\![\texttt{F}]\!]\colon [\![\texttt{SIGC}]\!] \to [\![\texttt{SIGB}]\!]$.

Now, if we have a structure `s : SIGB`, modelled by an a-functor

$$[\![\texttt{s}]\!]\colon [\![\texttt{SIGB}]\!] \longrightarrow \mathcal{S}$$

the structure `u = F(s)` of signature `SIGC` is interpreted as the composite

$$[\![\texttt{u}]\!] = [\![\texttt{s}]\!] \circ [\![\texttt{F}]\!]\colon [\![\texttt{SIGC}]\!] \longrightarrow [\![\texttt{SIGB}]\!] \longrightarrow \mathcal{S}$$

(Cf. the instantiation functors of 4.2.) So, for example,

$$[\![\texttt{u.ca}]\!] = [\![\texttt{s}]\!] \circ [\![\texttt{F}]\!] \circ [\![\texttt{ca}]\!] = [\![\texttt{s}]\!] \circ [\![\texttt{ba1}]\!] = [\![\texttt{s.ba1}]\!]$$

which is what we want.

4.4 Extended ML specifications are classifying toposes

It can be shown that $[\![\texttt{SIG}]\!]$ is always a *slice* topos of $\mathcal{S}$, i.e.

$$[\![\texttt{SIG}]\!] = \mathcal{S}/S$$

for some $S \in \mathcal{S}$. This S is (literally) the "set of structures of signature `SIG` in $\mathcal{S}$".

Suppose we adjoin some axioms to `SIG` to obtain an Extended ML signature `SIG'`. These axioms will define a subset (subobject)

$$S' \rightarrowtail S$$

the "set of structures of signature `SIG` in $\mathcal{S}$ satisyfing the axioms". Hence we get a new slice topos

$$[\![\texttt{SIG'}]\!] = \mathcal{S}/S' \hookrightarrow \mathcal{S}/S = [\![\texttt{SIG}]\!]$$

and it is not hard to show that this is the classifying topos for structures of signature `SIG'`, in an obvious sense: it contains the generic structure of signature `SIG` that satisfies the axioms, and, since a-functors preserve the truth of all set-theoretic assertions, any instantiation of this structure via an a-functor $[\![\texttt{SIG'}]\!] \to \mathcal{S}$ will also satisfy the axioms.

When extended to specifications, the polynomial category approach to ML polymorphism gives us a way of defining polymorphic predicates and formulate suitable polymorphic axioms with implicit type quantifiers. We simply work within the logic of the classifying topos, which is set-theoretic. This seems to be a useful way of interpreting, and reasoning about, polymorphic specifications—an aspect of Extended ML that has, up to now, seemed problematic.

5 Some final comments

This paper is a prospectus, not a report on work already completed. But the approach already sheds some light on the 'meaning' of Modules in SML and Extended ML, while retaining a pleasantly close relationship with the official Definition.

Our work does not use the full power of synthetic domain theory. We didn't need 'real' polymorphic types, upper, lower and convex powerdomains, or subtyping and bounded quantification. But the fact that these constructions exist in $\mathcal{E}\!f\!f$ means that our approach applies to languages more sophisticated than ML.

We have tried to show that ML is essentially a simple language, by interpreting it in an intuitively naive framework. We believe that many of the tools previously used to describe aspects of ML are too powerful: ML polymorphism is simpler than System F, and using type theory to model ML Modules is overkill. Both can be captured in a much more elementary way. It is also nice that the use of polynomial categories in both cases reflects the analogy between the *operational* treatments of ML polymorphism and ML Modules, emphasised in [2].

One attractive feature of the strategy sketched here is that it is modular. The interpretations of closed types, of first-order polymorphism, and of the Modules system can be kept quite separate. In fact, the last two make sense in a more general setting, namely that of *categories-with-structure*. However, in this case, classifying categories are constructed differently: cf. [17].

References

[1] P. Freyd, P. Mulry, G. Rosolini, D. S. Scott: Extensional PERs, in: Proc. of 5th Annual Symposium on Logic in Computer Science, 1990.

[2] R. Harper, R. Milner, M. Tofte: A type discipline for program modules, in: Proc. of TAPSOFT '87, Springer LNCS 250, 1987.

[3] R. Harper, J. Mitchell, E. Moggi: Higher-order modules and the phase distinction, in: Proc. 17th ACM Symposium on Principles of Programming Languages, 1990.

[4] J. M. E. Hyland: The effective topos, in: *The L. E. J. Brouwer Centenary Symposium* (ed. A. S. Troelstra, D. van Dalen), North-Holland, 1982.

[5] J. M. E. Hyland, E. P. Robinson, G. Rosolini: The discrete objects in the effective topos, *Proc. Lond. Math. Soc.* (3) 60 (1990) 1–36.

[6] P. T. Johnstone: *Topos Theory*, Academic Press, 1977.

[7] E. Kazmierczak: An institution for Extended ML, talk at the 9th Workshop on Abstract Data Types; report, in preparation.

[8] J. Lambek, P. J. Scott: *Introduction to Higher-Order Categorical Logic*, Cambridge University Press, 1986.

[9] D. MacQueen: Modules for Standard ML, in: *Standard ML* (R. Harper, D. MacQueen, R. Milner), available as ECS-LFCS-86-2, Laboratory for the Foundations of Computer Science, University of Edinburgh.

[10] D. MacQueen: Using dependent types to express modular structure, in: Proc. 13th ACM Symposium on Principles of Programming Languages, 1990.

[11] R. Milner, M. Tofte: *Commentary on Standard ML*, MIT Press, 1990.

[12] R. Milner, M. Tofte, R. Harper: *The Definition of Standard ML*, MIT Press, 1990.

[13] J. Mitchell, R. Harper: The essence of ML, in: Proc. 15th ACM Symposium on Principles of Programming Languages, 1988.

[14] J. Mitchell, G. D. Plotkin: Abstract data types have existential types, *ACM Trans. Programming Languages and Systems,* 10(3):470–502.

[15] W. K.-S. Phoa: Effective domains and intrinsic structure, in: Proc. of 5th Annual Symposium on Logic in Computer Science, 1990.

[16] W. K.-S. Phoa: *Domain Theory in Realizability Toposes*, dissertation, University of Cambridge, November 1990; available as CST-82-91/ECS-LFCS-91-171, Department of Computer Science, University of Edinburgh.

[17] W. K.-S. Phoa: A simple categorical model of first-order polymorphism, submitted.

[18] G. Rosolini: Categories and effective computations, in: Lecture Notes in Computer Science 283, Springer, 1987.

[19] D. T. Sannella, A. Tarlecki: Extended ML, an institution-independent framework for formal program development, in: *Proc. of Category Theory and Computer Programming (Guildford)*, Springer LNCS 240, 1987.

[20] R. D. Tennent, *Semantics of Programming Languages*, Prentice-Hall, 1991.

[21] M. Tofte: *Operational Semantics and Polymorphic Type Inference*, dissertation, 1988; available as CST-52-88/ECS-LFCS-88-54, Department of Computer Science, University of Edinburgh.